Uma Solução para o Atari Busy Police Game em Prolog

Víctor C. Colombo[1]

[1]Departamento de Computação - Universidade Federal de São Carlos (UFSCar)

vccolombo@tuta.io

Resumo. *Este trabalho propõe uma solução ao jogo de Atari Busy Police. Nesse jogo, é preciso ajudar o policial a chegar até um fugitivo localizado em um dado cenário. É então proposto um mecanismo de busca para encontrar um caminho acíclico, ou seja, sem repetição de estados, do policial ao fugitivo. Por fim é apresentada sua implementação em Prolog. É também mostrado como a adição de novas funcionalidades no jogo pode tornar o problema mais complexo.*

1. O Problema

O Busy Police Game funciona da seguinte forma: O cenário é composto de um malha, nesse caso de tamanho 10x5. O jogador inicia em uma posição pré-definida do cenário, normalmente no primeiro andar, e o fugitivo é inicializado em outra posição, normalmente no último andar. Para ganhar, o jogador deve prender o fugitivo, o que é definido como ambos estarem na mesma posição.

O jogador pode se movimentar pelo cenário seguindo algumas regras. Primeiramente, pode-se andar horizontalmente sem restrições, com excessão de quando há um obstáculo no caminho. Esse obstáculo é dado como um carrinho de supermercado no jogo. Para passar esse obstáculo, é necessário pulá-lo, ou seja, passar diretamente para a posição seguinte ao obstáculo. Esse movimento só é permitido caso não haja outro carrinho na posição seguinte, caso o fugitivo não esteja nessa posição, caso não haja uma parede (fim do cenário) ou caso não haja uma escada.

As escada são objetos especiais que levam o jogador de um andar para outro. Elas são bidirecionais, ou seja, podem levar o jogador tanto para cima quanto para baixo. Para passar de um andar para outro, é preciso estar na mesma posição que uma escada. Com essas definições, podemos então propor uma solução automatizada para encontrar o caminho da posição inicial do policial até a posição que o fugitivo se encontra.

2. A Solução

Para resolver esse problema, é proposta uma solução recursiva. O caso base é o nosso objetivo, a captura do fugitivo. Esse estado deve ser unificado e retornar $true$ apenas quando o policial e o fugitivo estiverem na mesma posição.

Em seguida, os casos recursivos foram definidos como as ações que o policial pode tomar para se movimentar no cenário. A primeira na ordem de prioridades é o uso da escada. Como o fugitivo costuma estar em andares superiores, é preferível subir o mais rápido possível o mapa do que mover apenas horizontalmente no começo. Essa regra é definida da seguinte forma: é verificado se há uma escada na posição atual do policial. Caso isso seja positivo, é chamada a recursão, com a nova posição sendo o andar superior. Analogamente é feito para descer as escadas. A ação de descer as escadas permite que cenários mais complexos sejam possíveis.

Em seguida, é feita a regra para pular um carrinho. Essa regra faz com que o policial pule duas posições para o lado, caso haja um carrinho em seu caminho. Para isso, é verificado a presença do carrinho, e também se não há nenhum obstáculo na nova posição a ser alcançada. Afinal, de acordo com as regras, o policial não pode pousar em um local em que haja o fugitivo, uma escada, outro carrinho, ou *out of bounds*. Caso todos esses requisitos sejam confirmados (unificados), é chamada a recursão.

Note aqui como é importante colocar sempre a regra recursiva no final. Isso garante que ela seja chamada apenas caso realmente seja possível unificar tudo na regra, o que garante desempenho e evita loops.

Finalmente, o mais trivial, o movimento horizontal. Devemos garantir apenas

que não haja obstáculos, como carrinhos ou paredes.

3. Uma Solução Acíclica

Para gerar uma solução acíclica e única, devemos encontrar uma forma de manter registro das ações que já tomamos. Para isso, fazemos uma lista, inicialmente vazia, de estados já visitados, e a cada estado visitado colocamos esse novo estado como cabeça da lista. No fim, teremos o caminho percorrído sem repetições de estados e facilmente retornável e legível em Prolog.

Isso também nos permite evitar que nosso policial fique andando em círculos. Basta checar se a posição atual já foi visitada a cada regra. Caso sim, não levamos a recursão a frente, e fazemos regressão. Afinal, se aquele estado já foi visitado, ou ele é inútil, ou seus próximos caminhos já estão na pilha de recursão para serem avaliados.

4. A Resposta

Podemos finalmente encontrar nossa resposta para os problemas proprostos no trabalho. Nossa regra é do tipo $consegue/4$, sendo os dois primeiros argumentos a posição do policial e do ladrão, respectivamente, e os dois últimos o caminho já percorrido (para fazer a checagem proposta na seção anterior) e o caminho percorrido até o fugitivo. Esse último terá todo o caminho que o policial deve percorrer, ou seja, a resposta pedida na etapa 2 desse trabalho.

Para encontrarmos a solução, devemos fazer uma *query* ao Prolog. Nessa solução ela deve ser do tipo *consegue(estado(X, Y), estado(Z, W), [], R)*, sendo X e Y a posição horizontal e vertical do policial, e Y e Z as do ladrão. R é o caminho que o policial deve percorrer para alcançar o fugitivo.

Caso seja possível alcançar o fugitivo, devemos ter como resposta $true$, e todo o caminho que o policial percorre. Nessa situação, é possível apertar *;* para procurar uma solução nova.

Caso não haja mais nenhuma solução possível, Prolog retornará $false$. Isso mostra que nossa solução evita loops, e sempre retorna uma resposta.

Abaixo encontra-se o código para solucionar o problema proposto, e os 4 cenários dados no trabalho são apresentados com possíveis respostas.

Listing 1. Código Solução do Game

```
obstaculo(X, Y) :- carrinho(X, Y); X < 1; X > 10.


% Captura o fugitivo
consegue(estado(Hor, Ver), estado(Hor, Ver), R, R).


% Escadas
consegue(estado(PHor, PVer), estado(LHor, LVer), SoFar, R) :-
  PVer_new is PVer + 1, escada(PHor, PVer),
  \+ memberchk( estado(PHor, PVer), SoFar ),
  consegue(estado(PHor, PVer_new), estado(LHor, LVer),
     [estado(PHor, PVer) | SoFar], R).
consegue(estado(PHor, PVer), estado(LHor, LVer), SoFar, R) :-
  PVer_new is PVer - 1, escada(PHor, PVer_new),
  \+ memberchk( estado(PHor, PVer), SoFar ),
  consegue(estado(PHor, PVer_new), estado(LHor, LVer),
     [estado(PHor, PVer) | SoFar], R).

% Pular Carrinho
consegue(estado(PHor, PVer), estado(LHor, LVer), SoFar, R) :-
   PHor_Next is PHor + 1, carrinho(PHor_Next, PVer),
   PHor_new is PHor + 2, PHor_new \= LHor,
      not(obstaculo(PHor_new, PVer)), not(escada(PHor_new,
      PVer)),
   \+ memberchk( estado(PHor, PVer), SoFar ),
   consegue(estado(PHor_new, PVer), estado(LHor, LVer),
      [estado(PHor, PVer) | SoFar], R).
```

```
24 consegue(estado(PHor, PVer), estado(LHor, LVer), SoFar, R) :-
25    PHor_Next is PHor - 1, carrinho(PHor_Next, PVer),
26    PHor_new is PHor - 2, PHor_new \= LHor,
         not(obstaculo(PHor_new, PVer)), not(escada(PHor_new,
         PVer)),
27    \+ memberchk( estado(PHor, PVer), SoFar ),
28    consegue(estado(PHor_new, PVer), estado(LHor, LVer),
         [estado(PHor, PVer) | SoFar], R).
29
30
31 % Mover Horizontal
32 consegue(estado(PHor, PVer), estado(LHor, LVer), SoFar, R) :-
33   PHor_new is PHor + 1, not(obstaculo(PHor_new, PVer)),
34   \+ memberchk( estado(PHor, PVer), SoFar ),
35   consegue(estado(PHor_new, PVer), estado(LHor, LVer),
        [estado(PHor, PVer) | SoFar], R).
36 consegue(estado(PHor, PVer), estado(LHor, LVer), SoFar, R) :-
37   PHor_new is PHor - 1, not(obstaculo(PHor_new, PVer)),
38   \+ memberchk( estado(PHor, PVer), SoFar ),
39   consegue(estado(PHor_new, PVer), estado(LHor, LVer),
        [estado(PHor, PVer) | SoFar], R).
```

A função memberchk verifica se um elemento faz parte de uma lista e retorna verdadeiro caso sim. O operador \+ é equivalente à negação.

Listing 2. Ambiente 1

```
1 carrinho(8, 1). % carrinho(Hor, Ver)
2 carrinho(3, 2).
3 carrinho(4, 3).
4 carrinho(3, 5).
5 carrinho(4, 5).
```

```
6
7  escada(2, 1). % escada(Hor, Ver)
8  escada(9, 1).
9  escada(5, 2).
10 escada(3, 3).
11 escada(8, 3).
12 escada(10, 3).
13 escada(1, 4).
14 escada(6, 4).
```

Listing 3. Resultado do Ambiente 1

```
1  ?- consegue(estado(10, 1), estado(10, 5), [], R).
2  R = [estado(9, 5), estado(8, 5), estado(7, 5), estado(6, 5),
      estado(6, 4), estado(7, 4), estado(8, 4), estado(8, 3),
      estado(..., ...)|...] [write]
3  R = [estado(9, 5), estado(8, 5), estado(7, 5), estado(6, 5),
      estado(6, 4), estado(7, 4), estado(8, 4), estado(8, 3),
      estado(7, 3), estado(6, 3), estado(5, 3), estado(5, 2),
      estado(6, 2), estado(7, 2), estado(8, 2), estado(9, 2),
      estado(9, 1), estado(10, 1)] .
```

Listing 4. Ambiente 2

```
1  carrinho(8, 1). % carrinho(Hor, Ver)
2  carrinho(3, 2).
3  carrinho(7, 3).
4  carrinho(3, 4).
5  carrinho(4, 5).
6
7  escada(2, 1). % escada(Hor, Ver)
```

```
escada(9, 1).
escada(10, 1).
escada(6, 2).
escada(1, 3).
escada(8, 3).
escada(1, 4).
escada(9, 4).
```

Listing 5. Resultado do Ambiente 2

```
?- consegue(estado(5, 1), estado(7, 5), [], R).
R = [estado(6, 5), estado(5, 5), estado(3, 5), estado(2, 5),
   estado(1, 5), estado(1, 4), estado(1, 3), estado(2, 3),
   estado(..., ...)|...] [write]
R = [estado(6, 5), estado(5, 5), estado(3, 5), estado(2, 5),
   estado(1, 5), estado(1, 4), estado(1, 3), estado(2, 3),
   estado(3, 3), estado(4, 3), estado(5, 3), estado(6, 3),
   estado(6, 2), estado(5, 2), estado(4, 2), estado(2, 2),
   estado(2, 1), estado(3, 1), estado(4, 1), estado(5, 1)] .
```

Listing 6. Ambiente 3

```
carrinho(3, 2). % carrinho(Hor, Ver)
carrinho(5, 2).
carrinho(7, 2).
carrinho(7, 3).
carrinho(8, 4).
carrinho(7, 5).

escada(9, 1). % escada(Hor, Ver)
escada(1, 2).
```

```
escada(10, 3).
escada(5, 4).
```

Listing 7. Resultado do Ambiente 3

```
?- consegue(estado(3, 1), estado(1, 5), [], R).
R = [estado(2, 5), estado(3, 5), estado(4, 5), estado(5, 5),
   estado(5, 4), estado(6, 4), estado(7, 4), estado(9, 4),
   estado(..., ...)|...] [write]
R = [estado(2, 5), estado(3, 5), estado(4, 5), estado(5, 5),
   estado(5, 4), estado(6, 4), estado(7, 4), estado(9, 4),
   estado(10, 4), estado(10, 3), estado(9, 3), estado(8, 3),
   estado(6, 3), estado(5, 3), estado(4, 3), estado(3, 3),
   estado(2, 3), estado(1, 3), estado(1, 2), estado(2, 2),
   estado(4, 2), estado(6, 2), estado(8, 2), estado(9, 2),
   estado(9, 1), estado(8, 1), estado(7, 1), estado(6, 1),
   estado(5, 1), estado(4, 1), estado(3, 1)] .
```

Listing 8. Ambiente 4

```
carrinho(7, 1). % carrinho(Hor, Ver)
carrinho(7, 2).
carrinho(7, 3).
carrinho(7, 4).
carrinho(7, 5).

escada(9, 1). % escada(Hor, Ver)
escada(2, 2).
escada(10, 3).
escada(4, 3).
escada(6, 4).
```

Listing 9. Resultado do Ambiente 4

```
1 ?- consegue(estado(3, 1), estado(10, 5), [], R).
2 R = [estado(9, 5), estado(8, 5), estado(6, 5), estado(6, 4),
     estado(5, 4), estado(4, 4), estado(4, 3), estado(3, 3),
     estado(..., ...)|...] [write]
3 R = [estado(9, 5), estado(8, 5), estado(6, 5), estado(6, 4),
     estado(5, 4), estado(4, 4), estado(4, 3), estado(3, 3),
     estado(2, 3), estado(2, 2), estado(3, 2), estado(4, 2),
     estado(5, 2), estado(6, 2), estado(8, 2), estado(9, 2),
     estado(9, 1), estado(8, 1), estado(6, 1), estado(5, 1),
     estado(4, 1), estado(3, 1)] .
```

5. Um Problema Adicional

É possível propor modificações ao jogo para torná-lo ainda mais desafiador. Uma técnica muito famosa em jogos é o fato de só se conseguir realizar uma ação após realizar outras ações anteriormente, como coletar itens ou conversar com outros personagens. Nesse trabalho, é proposto que o policial só deve poder realizar a prisão do fugitivo caso haja evidências de sua participação em um crime. Afinal, vivemos em um Estado de Direito e o fugitivo é inocente a não ser que haja provas que digam o contrário.

Contudo, nosso fugitivo não é muito inteligente e deixou cair diversas evidências no chão do cenário em sua fuga. Para prender o suspeito, o policial deve coletar todas as evidências que há no cenário antes de chegar até ele. Essas evidências são colocadas de forma programática, da mesma forma que as escadas e os carrinhos.

Outro detalhe importante é que o policial não pode pular um carrinho e pousar sobre uma evidência. Afinal, isso destruiria a evidência. Ele deve chegar até a evidência por meio de um movimento horizontal simples, e então coletar a prova caso ambos compartilhem a mesma posição. Ao coletar a evidência, ela é anexada a uma lista que mantém o registro de todas as evidências coletadas. Ao alcançar o fugitivo, é verificado se foram

coletadas todas as evidências antes de se efetuar a prisão.

Listing 10. Código Funcionalidade Extra

```
carrinho(7, 1). % carrinho(Hor, Ver)
carrinho(7, 2).
carrinho(7, 3).
carrinho(7, 4).
carrinho(7, 5).

escada(9, 1). % escada(Hor, Ver)
escada(2, 2).
escada(10, 3).
escada(4, 3).
escada(6, 4).

evidencia(1,5).
evidencia(1,4).
evidencia(1,3).

obstaculo(X, Y) :- carrinho(X, Y); X < 1; X > 10.


% Captura o fugitivo
consegue(estado(Hor, Ver, Ev), estado(Hor, Ver), R, R) :-
   length(Ev, 3).

% Evidências
consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R)
   :-
  evidencia(PHor, PVer), \+ memberchk(evidencia(PHor, PVer),
     Ev),
  \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
  consegue(estado(PHor, PVer, [evidencia(PHor, PVer) | Ev]),
```

```
     estado(LHor, Lver), [estado(PHor, PVer, Ev) | SoFar], R).

% Escadas
consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R) :-
  PVer_new is PVer + 1, escada(PHor, PVer),
  \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
  consegue(estado(PHor, PVer_new, Ev), estado(LHor, Lver),
     [estado(PHor, PVer, Ev) | SoFar], R).
consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R) :-
  PVer_new is PVer - 1, escada(PHor, PVer_new),
  \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
  consegue(estado(PHor, PVer_new, Ev), estado(LHor, Lver),
     [estado(PHor, PVer, Ev) | SoFar], R).

% Pular Carrinho
consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R) :-
   PHor_Next is PHor + 1, carrinho(PHor_Next, PVer),
   PHor_new is PHor + 2, PHor_new \= LHor,
      not(obstaculo(PHor_new, PVer)), not(escada(PHor_new,
      PVer)), not(evidencia(PHor_new, PVer)),
   \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
   consegue(estado(PHor_new, PVer, Ev), estado(LHor, Lver),
      [estado(PHor, PVer, Ev) | SoFar], R).
consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R) :-
   PHor_Next is PHor - 1, carrinho(PHor_Next, PVer),
   PHor_new is PHor - 2, PHor_new \= LHor,
      not(obstaculo(PHor_new, PVer)), not(escada(PHor_new,
      PVer)), not(evidencia(PHor_new, PVer)),
   \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
```

```
49    consegue(estado(PHor_new, PVer, Ev), estado(LHor, Lver),
          [estado(PHor, PVer, Ev) | SoFar], R).
50
51  % Mover Horizontal
52  consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R)
      :-
53    PHor_new is PHor + 1, not(obstaculo(PHor_new, PVer)),
54    \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
55    consegue(estado(PHor_new, PVer, Ev), estado(LHor, Lver),
          [estado(PHor, PVer, Ev) | SoFar], R).
56  consegue(estado(PHor, PVer, Ev), estado(LHor, Lver), SoFar, R)
      :-
57    PHor_new is PHor - 1, not(obstaculo(PHor_new, PVer)),
58    \+ memberchk( estado(PHor, PVer, Ev), SoFar ),
59    consegue(estado(PHor_new, PVer, Ev), estado(LHor, Lver),
          [estado(PHor, PVer, Ev) | SoFar], R).
```

O cenário é uma cópia do ambiente 4 proposto inicialmente no trabalho. Porém, dessa vez, com as evidências distribuídas pelo shopping virtual. A evidência nas linhas 13 a 15 é um fato definido da forma $evidencia(Horizontal, Vertical)$, indicando a posição da evidência no cenário.

Nas linhas 23 a 27, é escrita a regra para coletar essas evidências. Primeiramente é checado se há uma evidência na posição atual do policial, e se essa evidência ainda não foi coletada. Caso não fosse feita essa checagem, coletariamos evidências repetidas ou em posições aleatórias.

Em seguida, procede-se como as regras propostas originalmente, checando se o estado já foi visitado, e então chamando a recursão na linha 27. Nessa linha, a prova é adicionada a uma lista que mantém o registro das evidências, para ser checada no momento da prisão do fugitivo. Esse detalhe é a grande mudança nesse código. o fato $estado$ ganha um novo argumento para o policial, a lista de evidências coletadas, e se mantém idêntico para o fugitivo.

Outra mudança de destaque está na regra de pular carrinhos. Foi adicionada a checagem de se o lugar de 'pouso' do policial possui uma evidência. Como dito anteriormente, o policial não deve pousar sobre uma evidência para não destruíla.

Esse código é eficiente, capaz de achar todos os caminhos possíveis para se coletar todas as evidências e prender o suspeito, e ainda não entra em loops caso não haja caminho até uma evidência ou o suspeito.

Referências

[1] Nicoletti, M. C.(2003). Cartilha Prolog, A. São Carlos: EDUFSCar, 2003.

[2] http://www.cs.trincoll.edu/ ram/cpsc352/notes/prolog/factsrules.html

[3] http://www.learnprolognow.org/lpnpage.php?pageid=top